Moving the
Hazard Prediction and Assessment Capability
to a
Distributed, Portable Architecture

31 August 2002

Computational Sciences and Engineering Division

Moving the
Hazard Prediction and Assessment Capability
to a
Distributed, Portable Architecture

Ronald W. Lee

31 August 2002

CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATED TERMS

API	application programming interface
ASCII	American Standard Code for Information Interchange
CADRG	Compressed Arc Digitized Raster Graphics
CGI	Common Gateway Interface
CIB	Controlled Image Base
COM/DCOM	Component Object Model/Distributed Component Object Model
CORBA	Common Object Request Broker Architecture
DLL	dynamic link library
DOD	Department of Defense
DTRA	Defense Threat Reduction Agency
EJB	Enterprise JavaBeans
FTP	File Transfer Protocol
GIS	geographic information system
GUI	graphical user interface
HLA	High Level Architecture
HPAC	Hazard Prediction and Assessment Capability
HTML	Hypertext Markup Language
HTTP	Hypertext Transport Protocol
IDL	Interface Definition Language
IIOP	Internet Inter-Orb Protocol
ITPTS	Integrated Target Planning Toolset
J2EE	Java 2 Enterprise Edition
J2SE	Java 2 Standard Edition
JNDI	Java Naming and Directory Interface
JNLP	Java Network Launching Protocol
JVM	Java Virtual Machine
MFC	Microsoft Foundation Classes
NIMA	National Imagery and Mapping Agency
ORB	object request broker
RMI	[Java] Remote Method Invocation
SCIPUFF	Second-order Closure Integrated Puff Model
SOAP	Simple Object Access Protocol
T&D	transport and dispersion
TCP/IP	Transmission Control Protocol/Internet Protocol
UDDI	Universal Description Discovery and Integration
URL	Universal Resource Locator
VM	virtual machine
XML	Extensible Markup Language
WSDL	Web Services Description Language

ABSTRACT

The Hazard Prediction and Assessment Capability (HPAC) has been re-engineered from a Windows application with tight binding between computation and a graphical user interface (GUI) to a new distributed object architecture. The key goals of this new architecture are platform portability, extensibility, deployment flexibility, client-server operations, easy integration with other systems, and support for a new map-based GUI. Selection of Java as the development and runtime environment is the major factor in achieving each of the goals, platform portability in particular. Portability is further enforced by allowing only Java components in the client. Extensibility is achieved via Java's dynamic binding and class loading capabilities and a design by interface approach. HPAC supports deployment on a standalone host, as a heavy client in client-server mode with data stored on the client but calculations performed on the server host, and as a thin client with data and calculations on the server host. The principle architectural element supporting deployment flexibility is the use of Universal Resource Locators (URLs) for all file references. Java WebStart™ is used for thin client deployment. Although there were many choices for the object distribution mechanism, the Common Object Request Broker Architecture (CORBA) was chosen to support HPAC client server operation. HPAC complies with version 2.0 of the CORBA standard and does not assume support for pass-by-value method arguments. Execution in standalone mode is expedited by having most server objects run in the same process as client objects, thereby bypassing CORBA object transport. HPAC provides four levels for access by other tools and systems, starting with a Windows library providing transport and dispersion (T&D) calculations and output generation, detailed and more abstract sets of CORBA services, and reusable Java components.

1 INTRODUCTION

HPAC is a Defense Threat Reduction Agency (DTRA) tool for calculating atmospheric transport and dispersion of materials and assessing collateral effects. With hundreds of registered users, it is deployed in many military headquarters and field units as well as in civilian emergency response organizations. HPAC was originally developed as a standalone Windows application with a tight coupling between the user interface and the calculation engine. Understandably, the early stages of HPAC development were focused on scientific methodologies and correct results. Nonetheless, a demand for HPAC on other platforms and the need to integrate HPAC with many other tools and systems called for a new architecture. Consequently, HPAC has been re-engineered to a client-server, distributed object architecture readily portable to other platforms. Driving a new HPAC architecture are several system level requirements, including:

- *portability* to execute the user interface on virtually any desktop platform and perform calculations on server platforms,
- *extensibility* to allow addition of new capabilities to a deployed system,
- *deployment flexibility* for a range of situations and environments,
- support for *client-server operation* to leverage high performance server systems, and
- expedited *integration* of HPAC components in other systems.

Functional requirements call for "map-based" user interaction, which is also addressed in the new architecture.

1

1.1 PORTABILITY

With many HPAC user organizations representing a range of hardware and operating systems, portability has long been a goal for HPAC. Significant past efforts have attempted to solve this problem in a couple of ways. First was use of a cross platform development environment, MainWin from MainSoft, to build a Unix version. This proved unsatisfactory due to the time required to complete the port, which lagged a version behind the Windows implementation. HPAC source included C/C++ code accessing Microsoft Foundation Classes (MFC) as well as Fortran using Absoft features to call the Win32 Application Programming Interface (API) from Fortran code. Among the many issues porting to Unix were differences in Fortran compilers and handling of shared objects vs Windows dynamic link libraries (DLLs). Moreover, the resulting application did not behave, look, or feel as other applications on a Unix desktop, resulting in a very unpleasant user experience.

A less ambitious effort followed to port only the calculation engine to Unix. The client remained a Windows application, but invocations of calculation routines were replaced with a client call to a socket server wrapping the engine. A ported calculation engine allowed powerful Unix hardware to be used for computations. Emulation environments such as SoftWindows95 made possible execution of the client Win32 application under Unix without incurring the performance degradation of calculations under an emulated environment. This attempt at porting was more successful but also suffered from a slight lag in versions as well as compiler issues in porting the calculation engine code.

Costs of explicit porting and maintenance efforts made it clear that portability must be an architectural goal for HPAC.

1.2 EXTENSIBILITY

At the heart of HPAC is Second-order Closure Integrated Puff (SCIPUFF), a Lagrangian puff dispersion model developed by Titan System Corporation.[1–4] Detailed descriptions of material releases, meteorology, terrain, and other inputs are fed to SCIPUFF, which calculates the transport and dispersion and tracks material concentrations, depositions, and doses.

HPAC also includes incident source models which take parameterized descriptions of operational incidents and produce material releases suitable for input to SCIPUFF. Source models have been added from time to time over HPAC's history and will continue to be added as needed to address new requirements. Therefore, it is necessary for the HPAC architecture to provide a framework for plugging in new source models without requiring a system rebuild. In its original form, HPAC provided this extensibility via Win32 DLLs.

Another characteristic of HPAC development that calls for an extensible architecture is the number and geographic spread of the organizations comprising the development team, literally coast to coast. Such a collaborative effort is aided by a framework in which pluggable modules can be tested and operated independently of other modules.

1.3 DEPLOYMENT FLEXIBILITY

HPAC system requirements call for three basic modes of deployment:

- a standalone system with no network connection,
- a client-server arrangement with data available on the client but calculations performed on a server (heavy client), and

- a client-server environment with data downloaded to the client from the server and calculations performed on the server (thin client).

Thin client deployment has an additional requirement to be *Web-based*, or accessible via a Web browser with no prior installation of the application. The range of deployment options is an indication of the range of environments and settings in which HPAC must execute. Operational requirements call for HPAC to run on a laptop in the field as well as on a headquarters network.

1.4 CLIENT-SERVER OPERATION

Some organizations have investments in hardware providing computational capabilities exceeding that of most desktop computers. HPAC is computationally intensive in nature with source model, transport, and dispersion calculations. As mentioned above, a means of using server hardware for HPAC is needed. In addition to computational systems, many organizations rely upon servers for running work groups and enterprises for file and print sharing, and some have centrally managed configurations in which client workstations mount everything from a central server. Thus, HPAC's architecture must provide for client-server operation and deployment. Further, functionality available via services on the network is easily accessible to other tools and systems, aiding integration.

1.5 INTEGRATION

DTRA's Technology Development directorate is employing network technologies to link tools and systems into integrated environments.[5] In addition to Department of Defense (DOD) wide initiatives, such as the High Level Architecture (HLA)[6] for simulation environments, DTRA is fielding collaborative tool environments such as the Integrated Target Planning Toolset (ITPTS).[7] HPAC will be a part of ITPTS and multiple HLA federations as well as other environments (e.g., WARSIM[8]).

As a standalone Windows application, HPAC didn't lend itself to integration. Rather, almost every use of HPAC in another system required a custom solution. An HPAC more integrable with other systems is the original motivation behind the new architecture, as well as it's principle goal.

1.6 MAP-BASED INTERACTION

Events modeled in HPAC occur at a geographic location, whether real or nominal. Consequently, geographic or map-based representation of HPAC projects is desired. There are many possible solutions for map-based display and user interaction, ranging from simple map images to full geographic information system (GIS) capabilities. This new capability is a key aspect of the new architecture.

1.7 MOTIVATION

HPAC's original implementation toward a particular platform is common among computational tools. This is partly because tools and techniques used in advanced architectures have only recently become widely available. Technologies available at the end of calendar year 1999 when the re-engineering of HPAC began were highly effective in enabling an architecture to achieve the goals and requirements described above. The choices from available alternatives and elements of the new HPAC architecture are a useful case study for re-engineering efforts for other systems.

This paper begins with a brief overview of the principle components of HPAC, a necessary background for understanding architectural choices. Elements of the new architecture are described in terms of the goals they achieve. Where relevant, choices from available or competing technologies are explained, and remaining architectural issues are discussed.

2   HPAC OVERVIEW

SCIPUFF, the T&D engine, and incident source models have been identified as key components, but there are many other pieces to HPAC.

Weather data is critical to dispersion modeling, and HPAC has GUI and reader components for ingesting surface observations, upper air profiles, and forecast meteorology, as well as a historical climatology database. Effects modules for computing collateral human effects have computational components called from the dispersion engine as well as GUI control components.

HPAC's "material database" is a collection of files, one per material. In the original architecture, material files were read by the T&D engine as well as routines in the application GUI. The new architecture includes a material server that provides all components a single point of access to the repository of material files.

HPAC includes many GUI components for editing parameters and properties and controlling the dispersion calculation process. Source models themselves have a computational component for producing releases from incident descriptions and GUI components for editing incident descriptions. Added for the new architecture is a map display or representation of HPAC projects and calculation results.

Fig. 1 illustrates the transition to the new architecture. What was a single component (PC-SCIPUFF in Fig. 1) containing the T&D engine (i.e., SCIPUFF) as well as the GUI is now separated into several client and server components. The T&D engine is represented by the "HPACtool" box in the new architecture, described in Section 3.6.1.

Fig. 2 is a screen shot of the HPAC client GUI, named the *Project Editor*. Icons on the Incident Definition panel represent incident source models. Incidents defined for the project appear with other project objects in the horizontal Object Palette below the map display. Rendered on the map are icons representing releases as well as contour plots. In HPAC an *incident* is a notional description of an event, and a *release* is a detailed description of a dispersed material. Incidents produce one or more releases, and releases are fed as input to the T&D engine.

A typical sequence of events for a user begins with the definition of one or more incidents. For this, the user interacts with GUI components associated with incident source models. The models generate detailed release descriptions associated with each incident, and other GUI components allow the user to edit the generated releases. Using additional GUI components, the user will define the weather for the project and optionally specify calculation parameters. Next, the user requests a dispersion calculation and interacts with HPACtool (SCIPUFF) to view feedback and answer questions, also via GUI components. When the calculation completes, the user requests plots on a map display and/or storage of results data in American Standard Code for Information Interchange (ASCII) files in one of many available formats.

3   NEW HPAC ARCHITECTURE

All the system goals, portability, extensibility, deployment flexibility, client-server operation, and easy integration with other systems are met in the new architecture as described in this sec-
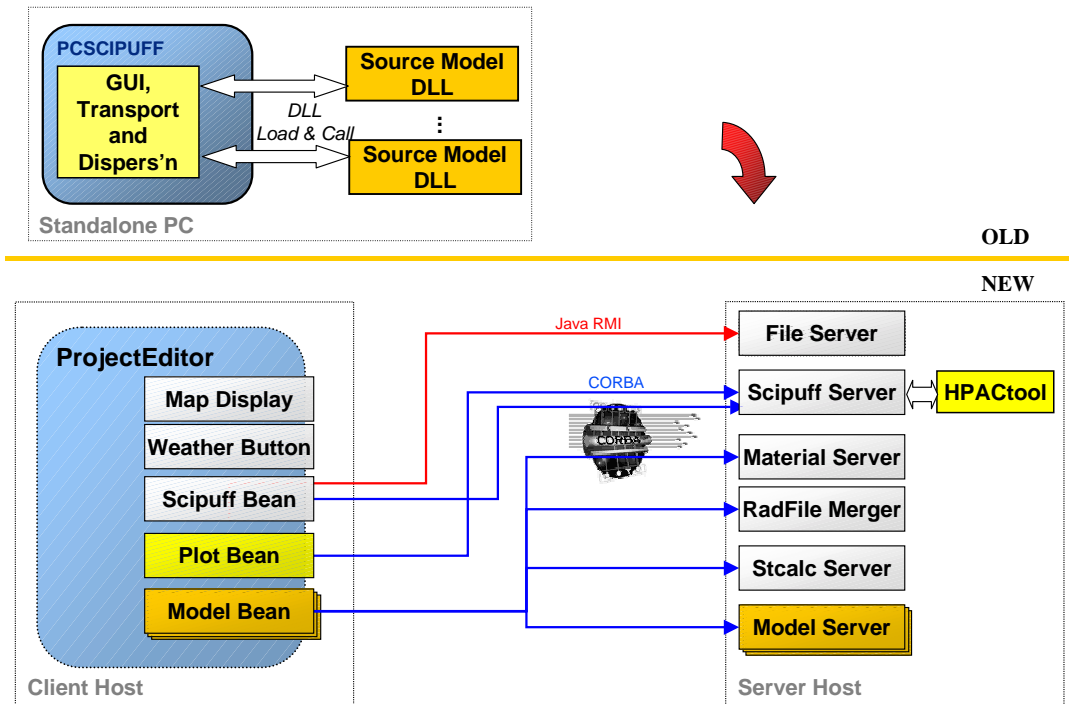
Fig. 1: Transition from the old to the new architecture.

tion. However, the kind of GUI, rich or Web-based, determines much of an application's architecture and was a key early decision for HPAC.

3.1 RICH GUI OR WEB APPLICATION

At the very beginning, HPAC's new architecture faced a critical decision. Any application involving user interaction and display has two choices for the means of providing the user experience: a user interface based on Hypertext Markup Language (HTML), or a *rich* user interface. An HTML interface implies a Web application with HTML pages served from a Web server and dynamic generation of the pages using one or more of a myriad of technologies ranging from Common Gateway Interface (CGI) scripts, to Java Servlets and Server Pages, to Enterprise Java Beans (EJB) hosted in a full Java 2 Enterprise Edition (J2EE) compliant application server. Regardless of the back end technologies used to generate Web pages, the user interacts with the application via a Web browser containing pages with HTML forms. The spartan capabilities of HTML form inputs can be augmented with Java applets or objects for plug-ins such as Macromedia Flash, but adding these elements moves the application toward the other end of the scale, or a rich user interface.
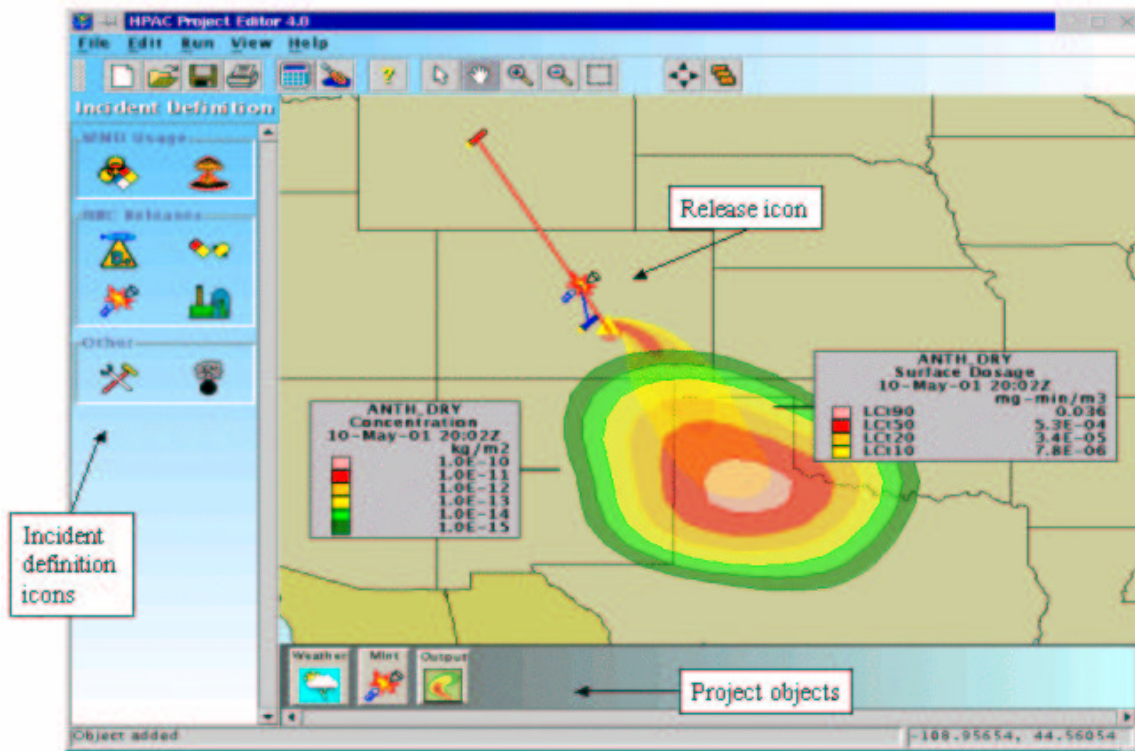
Fig. 2: New HPAC client GUI.

Ignoring hybrid approaches in which plug-in objects are embedded in an HTML page, a rich user interface involves a client application using some GUI framework and widget set. Advantages of a rich interface include:

- an improved user experience through more functional widgets and components,
- easier application maintenance,
- potentially lower up front development costs, and
- opportunities for reuse of interface components.[9]

A compelling argument in favor of HTML interfaces is the ease of deployment, for the user need only have a competent browser to access the application, regardless of the platform.[9] Further, user interactions are necessarily simplified if restricted to standard HTML form elements and the Web paradigm of page-based processing. However, the concept of rich user interfaces for applications driven from Web servers has received significant attention.[9,10]

For the new HPAC architecture, both Web-based HTML interfaces and a GUI application were considered. Requirements for very high levels of user interactions, such as dragging icons on a map to relocate releases, drove HPAC toward the use of a rich GUI. Section 3.2.1 describes how Java provides a rich, platform portable GUI for HPAC.

3.2   PORTABILITY

Among the many choices for the new architecture was the development language and runtime environment. HPAC in its original form was a combination of C, C++, and Fortran relying on

Win32 API calls and MFC for the GUIs. Several features of Java make it an easy choice for a platform portable system.

3.2.1 Java

Strictly speaking, there is a level of portability offered by C, C++, and Fortran, namely source code portability. However, variations among the compilers, in particular Fortran compilers, require modifications to the source. For example, some compilers place local variables on the stack by default, but some compilers make them static, making explicit `automatic` statements necessary. Regardless, source code portability requires recompilation for each target platform. The binding between source and platform-specific object code occurs at compile time and is therefore static.

Java portability differs in two principle ways, one *horizontal* and the other *vertical*. Java source is compiled to a platform-independent byte code which is executed within the Java Virtual Machine (JVM).[11] The specifics of the platform are addressed in the JVM implementation. Thus, the binding of the source code to the native object code occurs within the JVM at runtime. This is the *horizontal* dimension of Java portability.

A Java runtime environment includes many technologies and APIs, including GUI components, security, networking, distributed objects, naming services, and other functionality needed in HPAC. The Java editions and configurations define what core and extension technologies are available. For example, the Java 2 Standard Edition (J2SE) is the basis for desktop Java applications. All of the capabilities of J2SE are available with a J2SE implementation for a particular platform. This is the *vertical* dimension of Java portability.

As an illustration of the power of the vertical dimension, consider a platform portable application involving a GUI. For Windows, MFC and the Win32 API would be necessary, but for a Unix platform, X11 and Motif, Gtk, or Qt would be needed for a C/C++ GUI. With J2SE, this issue is solved via the abstract windowing toolkit and Java Foundation Classes which provide components for building a GUI. A component or application built with such components runs on any platform with a J2SE implementation.

Scripting environments (e.g., Tcl/Tk,[12,13] Python[14]) and cross platform GUI libraries such as the Fast Light Toolkit,[15] are alternatives to Java for achieving platform portability, but the latter address only one issue, GUIs. Scripting languages such as Python are intriguing and port to many platforms but do not include the range of technologies in J2SE. Further, scripting languages are weakly typed as opposed to Java's strong typing. Maintainability and robustness in weakly typed environments tend not to scale well.

True Object-Oriented Environment. In addition, Java is a true object-oriented environment.[16] C++ provides an object-oriented programming model, but the static binding of source to platform at compile time precludes transmission of an object across a network to a process running on a different platform, or the same platform where the code supporting the object is missing. Some argue Java is not a *pure* object-oriented language because it supports primitive types.[17] Nonetheless, it is *true* in the sense that source is not statically bound to a platform at compile time. Rather, transmission of an object, code and data, to a process on another platform is not only possible but is an integral part of the Java Remote Method Invocation (RMI) mechanism. As a true object-oriented development and runtime environment, Java supports a dynamic binding concept critical to the new HPAC architecture, as described in Section 3.3.1.

Java provides all of the characteristics of an object-oriented language: abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence.[18] The first five are commonly supported in modern languages, such as C++. Support for concurrency and persistence within the language and runtime environment is unique. Typically, these capabilities are achieved through operating system facilities, but Java supports them with language and environment constructs.

3.2.2   No Native Code on the Client

The layer of abstraction provided by the JVM comes with a performance cost. Although performance has been a focus of Java development with the HotSpot™ virtual machine (VM)[19] and has improved dramatically, Java cannot match native object code in execution speed.[20] Thus, native code, especially Fortran calculations, will remain in compute-intensive portions of HPAC.

However, migration to a client server architecture allows native code to be isolated at the server with only Java code at the client. HPAC enforces a "no native code on the client" rule in order to preserve portability of the GUI client application while accepting that supported server platforms will be limited to those for which the native C, C++, and Fortran code in server objects can be ported. This also limits the platforms for which a standalone deployment is possible but is a necessary compromise. Moreover, native code in the client application would have to be ported to all potential client platforms, thereby sacrificing the portability of the JVM layer and adding great complexity to a thin client deployment.

There are implications for this rule. Any client Java component accessing functionality implemented in native code must do so as a network client. All native functionality is provided as network services. As described in Section 3.5, functionality exported to external tools is provided in CORBA services, whereas functionality existing solely to support an HPAC Java client component may be provided using Java RMI.

Architectural elements supporting deployment flexibility (described in Section 3.4) also help meet the requirement for platform portability, but use of Java as the development and runtime environment provides most of what is necessary to meet the portability requirement.

3.3   EXTENSIBILITY

Design by interface, Java's support for dynamic class loading and reflection, and runtime configuration provide the basis for extensibility in the HPAC architecture. The principle applied concept is *dynamic binding*.

3.3.1   Dynamic Binding

Dynamic binding is a key attribute of an extensible, flexible system.[21] Whereas DLLs and shared objects are the mechanisms for dynamic binding with C, C++, and Fortran, Java provides dynamic class loading and reflection.

If the representation of the system definition and implementation, such as the source code, and the realization of that representation, such as a linked executable, are determined at compile and link time, the system is statically bound. Completely static binding in software is avoidable by reading configuration information at runtime, usually from configuration files. However, for languages compiled to native code, such as C++, there is no mechanism in the language to support dynamic or runtime binding. Rather, operating system mechanisms such as shared objects are the only means of achieving a dynamic binding of an implementation to an executing application.

8

Java's dynamic class loading mechanism is similar to DLLs, but the difference is the amount of a priori information necessary to make use of the dynamically loaded object. The functions or objects available in a dynamically loaded object must be known and planned for in the calling code. In contrast, this information can be discovered from a dynamically loaded Java class or object instance via the reflection mechanism.[22]

Moreover, a dynamically loaded Java object implementing a specific interface can be accessed and invoked in terms of that interface regardless of any additional capabilities or functionality it contains. Thus, dynamic class loading and design by interface in combination are the basis of a framework for pluggable components.

3.3.2 Design by Interface

Design by interface is a common approach for decoupling specification and implementation and is used throughout HPAC.[23,24] Often an abstract base class implementing some or all of the interface as well as providing additional support methods is provided. This follows the Template Method design pattern[25] and helps standardize behavior and provides commonly used functionality for reuse by concrete class implementations.

Although there are several such pluggable frameworks in HPAC, three are particularly illustrative of the concept as employed in HPAC: incident source model servers, incident source model beans, and map displays.

Incident Source Model Servers.   As described in Section 1.2 incident source models are critical components of HPAC for which an extensible framework is needed. Section 3.5.1 describes the choice of CORBA as the object distribution mechanism for exported services, and each incident model server provides such a CORBA service. Thus, the interface in this case is specified with CORBA Interface Definition Language (IDL), specifically the `IncidentModelServer` interface, represented in an abbreviated class diagram in Fig. 3.

Model servers have two client audiences, external tools needing the model's calculation and release-generating capabilities, and the T&D engine itself, which makes calls to the models during the calculation. `IncidentModelServer` defines methods for both these situations. The collaboration is illustrated in Fig. 4.

For this framework, the `IncidentModelServer` IDL interface defines how the other components of HPAC interact with the service. Each source model defines its own IDL interface extending `IncidentModelServer` and providing any additional capabilities as appropriate for the model. These additional methods are accessible to any client that accesses the server in terms of the model's defined interface.

`IncidentModelServerImpl` is the abstract base class providing default implementations of several methods. Note in Fig. 3 `XxxServerImpl` implements only those methods `IncidentModelServerImpl` does not provide via inheritance. This does not preclude any model server from providing a custom implementation of such a method, but it provides default behavior any model can reuse.

Use of the Template Method design pattern is demonstrated in the collaborations between the T&D engine and the source models, as shown in Fig. 4. While calculating, the engine calls a source model's `updateRelease()` method to update a release definition. This method is implemented in the abstract base class to perform some pre-processing, call another method, `updateReleaseData()`, and then perform necessary post-processing. All model servers must either

9
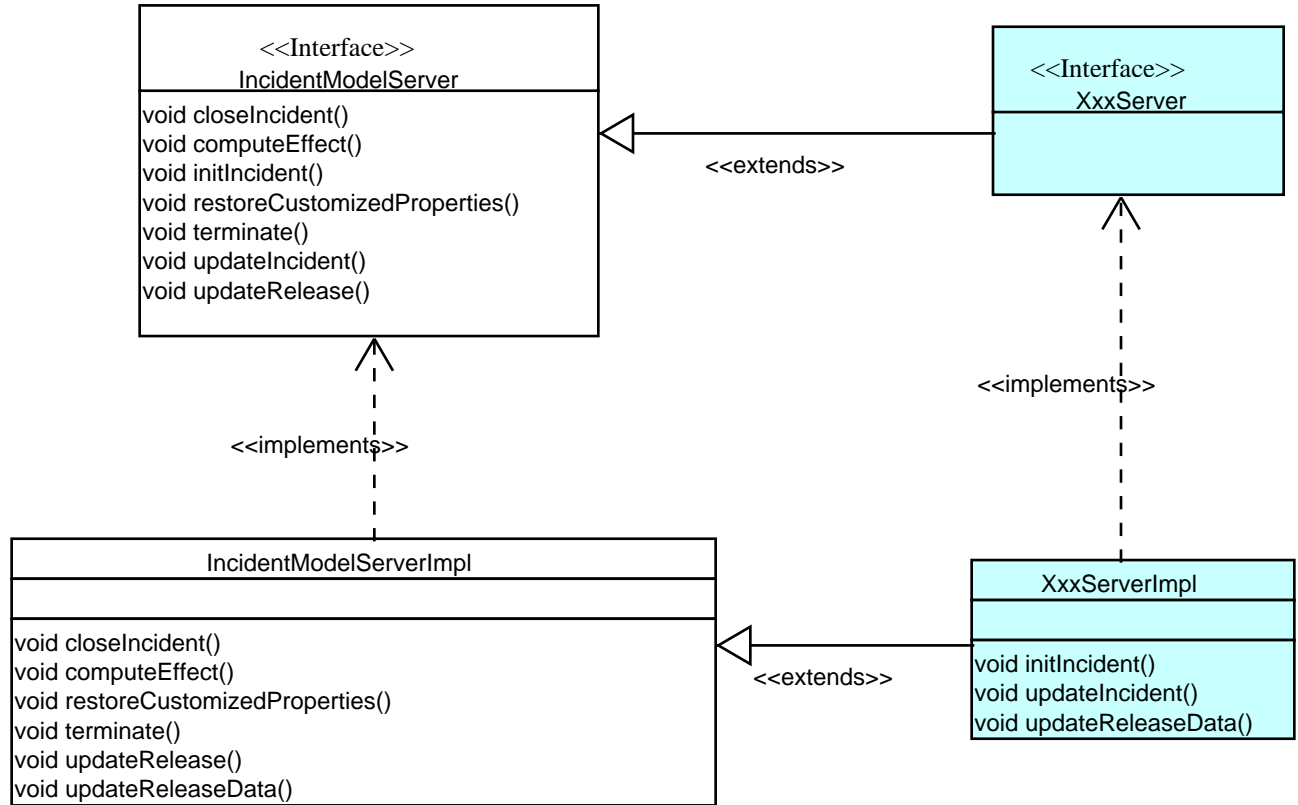
Fig. 3: IncidentModelServer framework.

implement `updateReleaseData()` for standard processing or override `updateRelease()` if non-standard processing is necessary. This pattern of specification by interface with implementation of standard behavior in an abstract base class is used throughout HPAC, both for client and server components.

Configuration files read at runtime inform the HPAC's server launcher component which incident source models are available, and each one identified is dynamically loaded and instantiated during system initialization. Snippets from the server configuration file follow:

```
hpacserver.servers.0=\
ScipuffFactory,\
mil.dtra.hpac.server.scipuff.impl.ScipuffServerFactoryImpl

hpacserver.servers.1=\
MaterialServer,\
mil.dtra.hpac.material.server.impl.MaterialServerImpl
...
hpacserver.servers.3=\
ChemBioWeaponFactory,\
mil.dtra.hpac.models.cbwpn.server.impl.ChemBioWeaponServerFactoryImpl

hpacserver.servers.4=\
SmokeWeaponFactory,\
```
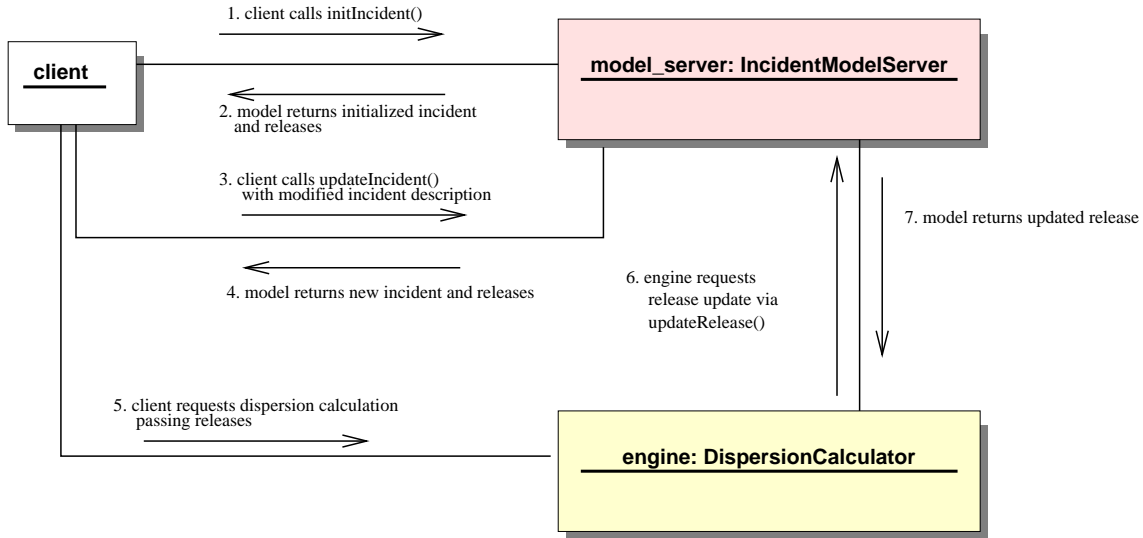
Fig. 4: Model server collaboration.

```
mil.dtra.hpac.models.swpn.server.impl.SmokeWeaponServerFactoryImpl
...
```

Each server is specified with a service name and the path of the implementation class to be dynamically loaded. Available servers are determined once at startup. Were the determination of which server class to load delayed until invocation, the binding would be even more dynamic and would support addition of model servers to a running system. However, a runtime deployment would result in additional delay for the load when a new model server is invoked the first time. For HPAC, any organization may provide their own model server implementation and add it to the configuration at startup.

Incident Source Model Beans. HPAC includes a Java client GUI, the Project Editor, for accessing HPAC services and managing projects. Users define and edit incidents via client side components (or JavaBeans™) provided by the incident source models. Beans are merely objects which adhere to a defined pattern. Specifically, object properties are exposed via *getter* and *setter* accessor methods whose names identify the property. For example, `getLocation()` and `setLocation()` are accessor methods for a *location* property.

Fig. 5 gives an abbreviated class diagram for incident model beans. Much of the required functionality is provided in abstract base classes `IncidentModel` and `ModelPanel`, representing the model bean and it's GUI editor bean, respectively.

Although a model bean is free to override any `IncidentModel` behavior, it is required to provide only one method, `createIncidentObjects()`, and should provide overrides for the serialization methods described in Section 3.3.3. Similarly, a model's extension of the `ModelPanel` base class need only provide three methods, `init()`, `load()`, and `store()`. For the `init()`, the model implementation may invoke `super.init()` to create an editor with standard tabs and controls or customize the appearance completely.

As with model servers, available model beans are specified via a runtime configuration file, and the configuration is processed only once, at startup. The snippet below is from the client

11

application properties or configuration file. Available models are specified as a list of model bean implementation class paths.

```
hpac.modelBeans=\
mil.dtra.hpac.client.models.Advanced,\
mil.dtra.hpac.models.cbwpn.client.ChemBioWeapon,\
mil.dtra.hpac.models.CBFac.client.CBFacWeapon,\
mil.dtra.hpac.models.swpn.client.SmokeWeapon,\
mil.dtra.hpac.models.mint.client.MissileIntercept,\
mil.dtra.hpac.models.nwi.client.NWI,\
mil.dtra.hpac.models.nfac.client.Nfac,\
mil.dtra.hpac.models.nwpn.client.Nwpn,\
mil.dtra.hpac.models.rwpn.client.RWPN
```

An organization may provide their own client model beans corresponding to their in-house model server implementations and configure HPAC's Project Editor to make them available. Regardless of what additional functionality any model bean provides, Project Editor dynamically loads the class specified in the configuration and instantiates objects, interacting with those objects solely in terms of the `IncidentModel` and `ModelPanel` abstract base classes.
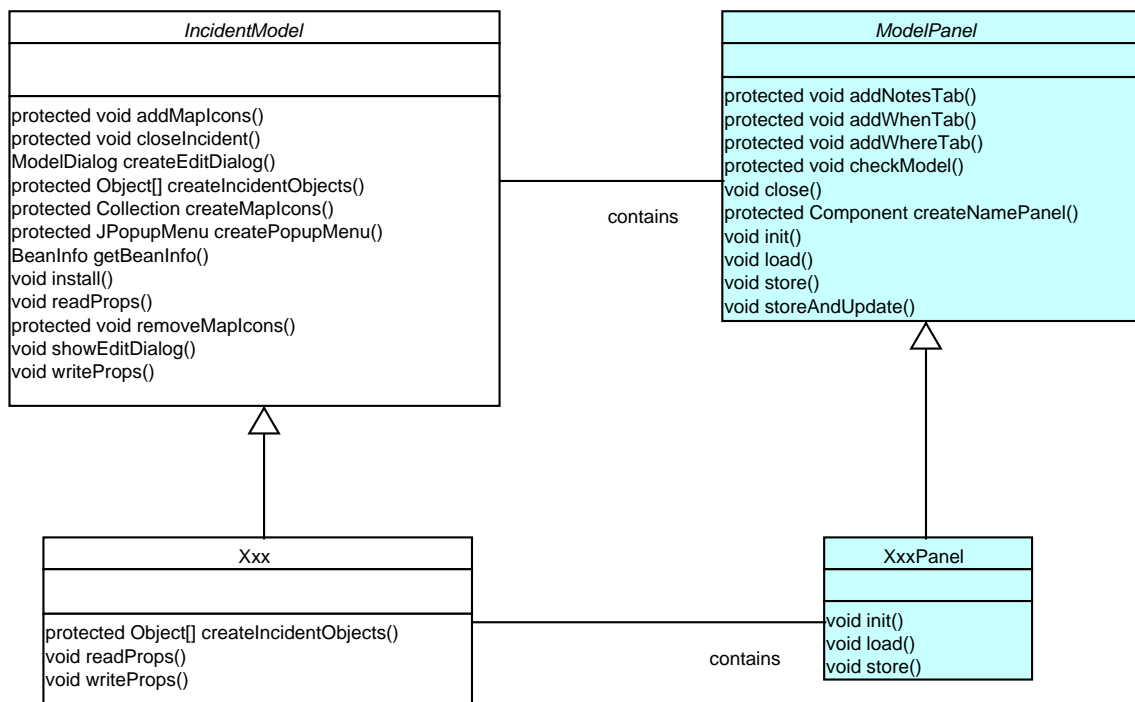
Fig. 5: IncidentModel bean framework.

Map Display.   Map display in an application can range from a simple generated image on top of which are drawn icons and other annotations to an environment in which the map display *is* the application, such as a GIS. In HPAC the map display is merely another GUI component or bean contained in the Project Editor window. Map display requirements call for many formats (e.g.,

National Imagery and Mapping Agency's [NIMA] Compressed Arc Digitized Raster Graphics [CADRG], Controlled Image Base [CIB] imagery, shapefiles) and data types (e.g., meteorology, terrain, roads, railroads, weather stations), with user control of the display.

The system and user requirements are met with OpenMap™[26] , which provides a map display bean. OpenMap provides extensibility through a layer framework, and organizations can implement layers and add them to the configuration dynamically. However, HPAC avoids a dependence on OpenMap and preserves the flexibility to use an open source or commercial product in the future. The simple map display framework illustrated in Fig. 6 accomplishes this.
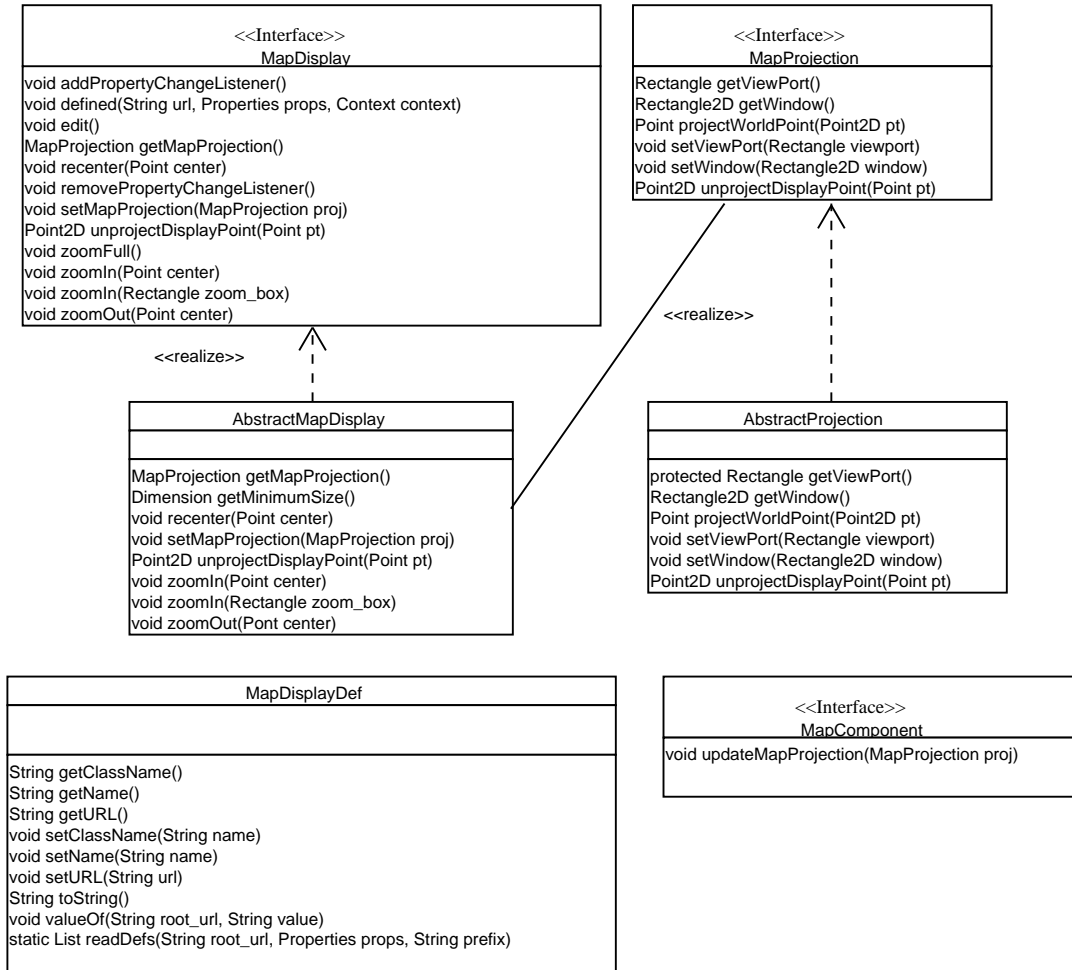
Fig. 6: Map display framework.

Once again the framework relies on a design by interface approach and runtime configuration and deployment of objects implementing the framework. Two interfaces define the framework, `MapDisplay` and `MapProjection`, and an abstract base class providing partial implementation of both is provided in `AbstractMapDisplay` and `AbstractMapProjection`, respectively. Provided with HPAC are implementations for OpenMap and a simpler implementation for display of an image referenced to geographic bounds. The following excerpt from the client application configuration file defines maps in the default HPAC installation.

```
hpac.maps.0=Global Land/Sea,mil.dtra.map.openmap.OmMapDisplay,\
${hpac.rootURL}/data/maps/landsea/landsea.map
hpac.maps.1=Global Political,mil.dtra.map.openmap.OmMapDisplay,\
${hpac.rootURL}/data/maps/admin/admin.map
```

The class path of the `MapDisplay` implementation is `mil.dtra.map.openmap.OmMapDisplay` for both map definitions. The URL specified for each map identifies an optional configuration file specific to the map display implementation. In this case, it's an OpenMap configuration file. A snippet from `admin.map` follows:

```
openmap.Projection=cadrg
openmap.layers=graticule DayNight contrast scaled
openmap.startUpLayers=scaled

contrast.class=mil.dtra.map.openmap.ContrastLayer
contrast.prettyName=Contrast Mask
contrast.color=80ffffff
DayNight.class=com.bbn.openmap.layer.daynight.DayNightLayer
DayNight.prettyName=Day/Night Areas
```

An organization deploying HPAC can customize maps at three levels. First, they can modify the respective OpenMap configurations of the identified maps. Here they may change parameters of provided layers and other configuration properties, or they can include their own OpenMap layer implementations. Note in the example above the `.class` property for a named layer identifies the class which OpenMap dynamically loads and instantiates. It is only necessary for the class to extend the `com.bbn.openmap.Layer` class.

Second, they may modify the HPAC application properties to specify other or additional maps (`hpac.maps.n` property) using the `mil.dtra.map.openmap.OmMapDisplay` implementation of the HPAC map display framework.

Third, they may provide their own implementation of the map display framework by naming a class that implements the `mil.dtra.map.MapDisplay` interface in a map display definition property. Note the HPAC map framework dynamically loads the specified class, `mil.dtra.map.openmap.OmMapDisplay` in the examples above.

3.3.3 Serialization via Java Properties

Like most applications, HPAC needs to save and load projects. An HPAC project is merely an object containing other objects comprising the state of the project as defined and edited by the user. In object-oriented terminology, saving an external representation of an object is referred to as object *persistence*, and the process of writing and reading an object's representation is called *serialization* and *deserialization*. (Reading and writing are often referred to collectively as *serialization*.) Rather than have the top level object, in this case the HPAC project, responsible for saving and loading a representation of all its contents, each component object shares the responsibility by (de)serializing its own state. Java provides binary object serialization for classes tagged with the `java.io.Serializable` interface, but a binary representation requires Java code to view or edit the serialized state of the project.

HPAC project files need to be human readable and suited for manipulation by other tools, including text editors. Further, HPAC needs a serialization mechanism that, like Java's binary serialization, is extensible and allows objects to self-serialize. The solution is a framework based

on Java properties files and the `java.util.Properties` class. An abbreviated class diagram is shown in Fig. 7.


```
┌─────────────────────────────────────────────────┐
│                  <<Interface>>                    │
│                  PropsSerializer                  │
├─────────────────────────────────────────────────┤
│ void readProps(ValueProperties props, String prefix)  │
│ void writeProps(ValueProperties props, String prefix) │
└─────────────────────────────────────────────────┘

                    <<realize>>

┌─────────────────────────────────────────────────┐
│              AbstractPropsSerializer              │
├─────────────────────────────────────────────────┤
│                                                   │
├─────────────────────────────────────────────────┤
│ void readProps(ValueProperties props, String prefix)  │
│ void writeProps(ValueProperties props, String prefix) │
└─────────────────────────────────────────────────┘

┌──────────────────────────────────────────────────────────────────┐
│                          <<Interface>>                             │
│                             Parser                                 │
├──────────────────────────────────────────────────────────────────┤
│ Class getObjectClass()                                             │
│ Object read(ValueProperties props, String prefix)                 │
│ boolean write(ValueProperties props, String prefix, Object object)│
└──────────────────────────────────────────────────────────────────┘

            0..*                        <<realize>>

                            ┌──────────────────────────┐
                            │       AbstractParser       │
                            ├──────────────────────────┤
                            │                            │
                            ├──────────────────────────┤
                            │ Class getObjectClass()     │
                            └──────────────────────────┘
```

```
┌──────────────────────────────────────────────────────────────────┐
│                          ValueProperties                           │
├──────────────────────────────────────────────────────────────────┤
│                                                                    │
├──────────────────────────────────────────────────────────────────┤
│ void addParsers(Parser[] parsers)                                  │
│ Object getArray(String key, Class type)                            │
│ boolean getBoolean(String key, boolean default)                    │
│ Object getDelimitedArray(String key, Class type, String delimiter) │
│ double getDouble(String key, double default)                       │
│ float getFloat(String key, float default)                          │
│ int getInt(String key, int default)                                │
│ Object getObject(String key, Class type, Object default)           │
│ PropsSerializer getObjectInstance(String key, Class type)          │
│ String getProperty(String key)                                     │
│ boolean putArray(String key, Object value)                         │
│ boolean putBoolean(String key, boolean value)                      │
│ boolean putDelimitedArray(String key, Object value, String delimiter)│
│ boolean putDouble(String key, double value)                        │
│ boolean putFloat(String key, float value)                          │
│ boolean putInt(String key, int value)                              │
│ boolean putObject(String key, Object value)                        │
│ void putObjectInstance(String key, PropsSerializer object)         │
│ void putObjectProperties(Object object, String prefix)             │
│ void removeParser(Parser parser)                                   │
│ void setObjectProperties(Object object, String prefix)             │
└──────────────────────────────────────────────────────────────────┘
```
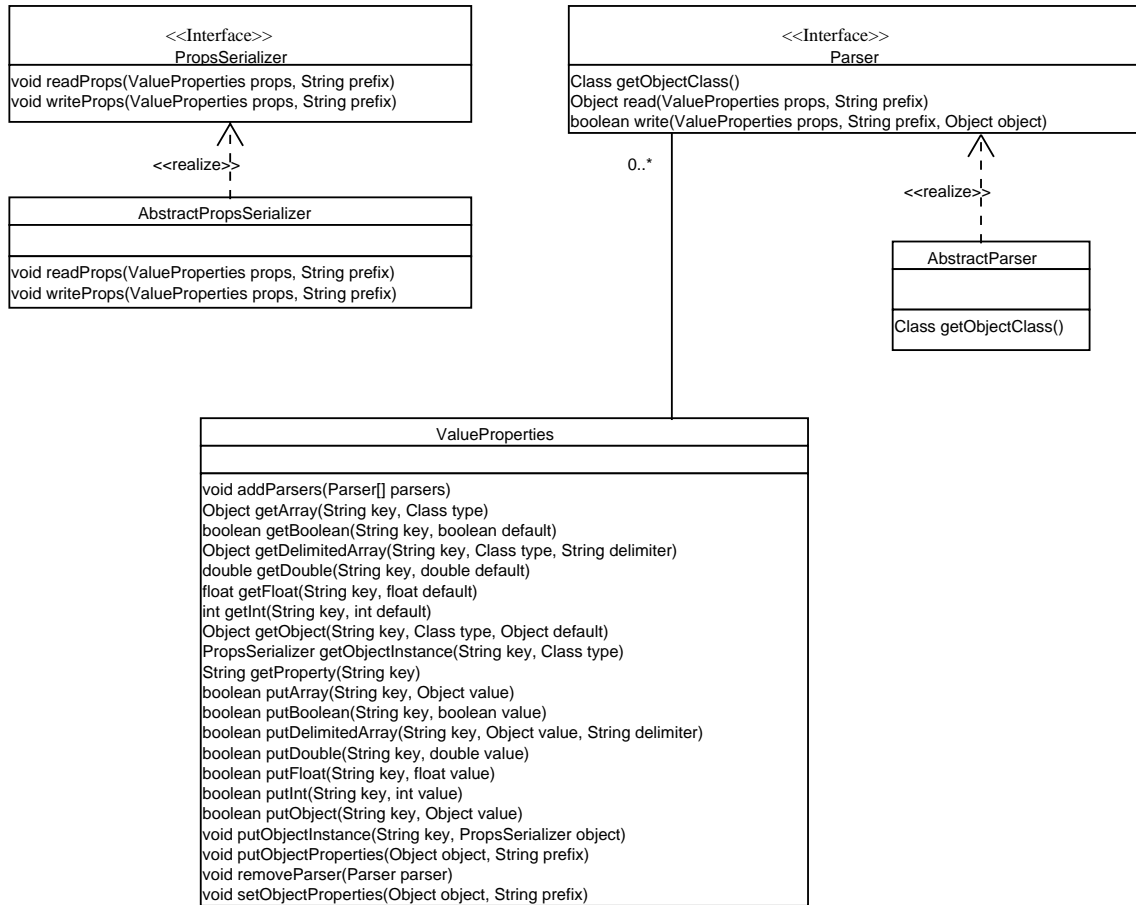
Fig. 7: Properties serialization framework.

Most of the implementation details are encapsulated in the `ValueProperties` class, which relies upon the `java.beans.Introspector` to determine an object's properties by referencing a `BeanInfo` provided for the object or using Java's reflection mechanism.

An object's properties are converted to a string representation in one of two ways. First, an object may be represented by a single string. In this case a `Parser` implementation is necessary and must be registered for the class in a `ValueProperties` object. `ValueProperties` includes parsers for common Java core classes such as `java.awt.Dimension` and `java.awt.Point`. A snippet from an HPAC project file follows.

```
URL=file\:/home/re7/.hpactest/nfac1.hpac
class=mil.dtra.hpac.data.Project
limits.class=mil.dtra.hpac.data.project.Limits
limits.maxGridCellsPerSurface=25000
limits.maxMetHorzSize=1000
limits.maxPuffs=20000
maxTimeStep=900.0
objectSet.0.class=mil.dtra.hpac.models.nfac.client.Nfac
```

15

```
objectSet.0.incident.ID=incident-1013192452738
objectSet.0.incident.class=mil.dtra.hpac.data.Incident
objectSet.0.incident.coord=-80.43389129638672,40.621944427490234
objectSet.0.incident.hasCustomMaterials=false
objectSet.0.incident.hasCustomReleases=false
objectSet.0.incident.location.class=mil.dtra.hpac.data.LLALocation
objectSet.0.incident.location.value=-80.43,40.62,0.0
```

Single string representation works well for classes with a small number of properties that are of primitive types like `int`, `boolean`, or `double`. Refer to the "objectSet.0.incident.coord" and "objectSet.0.incident.location.value" lines above. Properties of these objects are delimited with commas in a single value.

Second, an object may be represented as a hierarchy of properties and sub-objects with each leaf property stored as a single string. This is illustrated above by the limits and "objectSet. 0. incident" objects in the listing above. The limits object has three properties, *maxGridCellsPerSurface*, *maxMetHorSize*, and *maxPuffs*, each of type `int` and represented as a separate key or line in the file. A period separates the property names. For example, "limits. maxGridCellsPerSurface" represents the *maxGridCellsPerSurface* property of the *limits* property of the project object represented by the listing.

The *incident* property is an example of a hierarchical object. Of the five properties illustrated above, three are of primitive types: *ID*, *hasCustomMaterials*, and *hasCustomReleases*. The other two properties, *coord* and *location*, are themselves objects. Finally, the project object itself is a hierarchical object. In the listing we see its *URL*, *limits*, *maxTimeStep*, and *objectSet* properties.

In the framework described in Fig. 7, an object capable of serializing itself must implement the `PropsSerializer` interface, analogous to implementing `Serializable` for Java binary serialization. `PropsSerializer` specifies two methods, `readProps()` for deserialization and `writeProps()` for serialization. Hierarchical objects must implement `PropsSerializer`. Further, there are two serialization modes in this framework, *implicit* and *explicit*.

Implicit serialization relies on introspection to recognize the properties of an object, either via public getter and setter accessor methods as per the JavaBean pattern or a `BeanInfo` associated with the object. The return or parameter type class specified in the accessor is checked first to see if it implements `PropsSerializer`, in which case the parameter's `readProps()` or return value's `writeProps()` method is called for deserialization or serialization, respectively. Otherwise, the class is checked against registered parsers. If a match is found the parser is used to convert the object to or from a string representation. In the absence of a parser, the class is compared against the primitive types which are handled internally. Finally, the object is ignored if no information about its type can be determined.

Explicit serialization relies upon the class implementer to make explicit calls in `readProps()` and `writeProps()` methods to deserialize or serialize the object's properties. This is analogous to providing `readObject()` and `writeObject()` methods for binary serialization. A class with no other superclasses can extend `AbstractPropsSerializer` and inherit implicit serialization. Any class can invoke the implicit mechanisms by calling the `setObjectProperties()` and `putObjectProperties()` methods of the `ValueProperties` instance for implicit deserialization and serialization, respectively.

Such a framework is necessary for HPAC to be extensible. Any organization desiring to provide their own source model implementation must have a means of serializing properties of an incident description as part of the HPAC project. They can do so by providing implementations of the `PropsSerializer` methods `readProps()` and `writeProps()` for their `IncidentModel` ex-

16

tension. Other kinds of objects may be plugged into the serialization framework with implementations of `Parser` registered for their classes or `PropsSerializer` implementations with implicit or explicit serialization.

3.3.4   Jar Tiers and the Development Process

In C/C++ development, header files and libraries are the currency for exchanging software specifications and implementations, respectively. For Java, the medium of exchange is the Java archive or jar file, used for both development and execution. HPAC Java packages are organized into jars, and jars are organized into tiers of dependency. For the most part, the resulting jar organization reflects the structure of the development team, but since responsibility for various functionality was delegated among the developer organizations, this organization resulted in a reasonably functional packaging.

Jars are grouped into tiers, a tier representing a set of jars depending only on jars at lower tiers. Clearly, design changes in jars at lower tiers are more costly, cascading their effects to the higher tiers. Identification of dependency tiers is necessary in order to effectively manage changes affecting dependent code and requiring recompilation of dependent source. Examples of such changes include modification of a method signature or constant value and the removal of a class, interface, or method.

3.4   DEPLOYMENT FLEXIBILITY

As mentioned above, a key goal for the HPAC architecture is support for the range of deployment options dictated by the system requirements. Once again, core Java capabilities prove extremely helpful, in particular Java's networking (`java.net`) and I/O (`java.io`) packages and support for URLs[27] in identifying data sources.

3.4.1   URLs for File Reference

URLs specify a protocol for an address specification. URLs are commonly used to specify a resource available via the Hypertext Transport Protocol (HTTP), the `http:` protocol tag. URLs with a `file:` protocol simply specify a file on a filesystem accessible to the local host. The Java networking and I/O capabilities hide the specifics of the source of data and allow the using code to perform I/O via a single API, regardless of the location and mechanism whereby the data are transferred. A properly initialized `java.net.URL` object is used to retrieve a `java.io.InputStream` instance via via the `openStream()` method.

However, there is a limitation. An `InputStream` may be used to access data sequentially, not randomly. For configuration files, reference data, and other static, sequentially read data in HPAC, URLs are the *address* or locator of the source of the data. This allows HPAC to be configured to access this data via a local or remotely mounted filesystem using a `file:` URL, or the data may be placed behind a Web server and specified using `http:` URLs. Other URL protocols are possible, such as `ftp:` for the File Transfer Protocol (FTP), but Web service is preferred for reasons of security and access control. Note that serving static, sequential data from a Web server is a necessary prerequisite to deploying HPAC as a thin client, described in Section 3.5.5.

URLs work very well for sequentially accessed files, but direct- or random-access files cannot be read from or written to a URL. Access to random-access data must be brokered by a server object. If the size of the data is relatively small, a class providing the same methods as `java.io.RandomAccessFile` but actually reading and writing data across a network connection may be

used. A client class and corresponding RMI server are maintained to support that need, but the size of the data to be processed by HPAC components has precluded their use.

Two HPAC components in particular rely heavily on direct-access files: weather plotting, and map display. In both cases, the volume of data is too large to simply perform `RandomAccessFile` operations across the network. Rather, a service must be defined to send only the necessary data to the client. For weather plotting, these data would be contour polygons, marker positions, and such. For map displays, a map image would be produced for the client.

3.4.2 Runtime Configuration

Flexibility in deployment requires reading a dynamic, runtime configuration. HPAC uses system properties and properties files read into `java.util.Properties` objects to specify the system configuration. Examples have been provided above. This is also the means by which implementations of interfaces defined in the frameworks described in Section 3.3.2 are specified for the system.

3.5 CLIENT SERVER OPERATION

There are many ways to implement a client-server system using Transmission Control Protocol and Internet Protocol (TCP/IP) data and network layer protocols, respectively, which fuel the Internet.[28] Mechanisms range from socket-based communication to distributed object systems. It was clear early on that HPAC needed the latter, but there are several distributed object technologies available.

3.5.1 Choices for the Distributed Object Mechanism

Two technologies dominated early design discussions for HPAC: CORBA and Microsoft's Component Object Model (and Distributed Component Object Model, or COM / DCOM). Although DCOM has advantages for Windows platforms, it was eliminated early since it has not been ported to very many platforms, Solaris being virtually the only non-Windows platform supported.[29] Before settling on CORBA, some other technologies were explored, and the options within CORBA itself were investigated. Services written directly against sockets were also rejected early on as requiring too much attention to network communication and the necessity of establishing data exchange protocols. For all intents and purposes, CORBA was the only choice to meet the requirement to support non-Java clients running on any platform.

Java RMI.   For a pure Java application, RMI is the most effective and efficient means of deploying object-based services. With a Java client, the temptation to use RMI in HPAC was strong. However, RMI requires Java at both ends, which means non-Java clients would have to build a bridge between native code and a Java component to access the services. Although bridging products are available for specific technologies,[30] this was viewed as too contrary to HPAC system goals. Thus, RMI was rejected for services published to external tools but is used for "private" internal services supporting client components in the HPAC GUI.

Enterprise JavaBeans.   Another technology growing in the Java world at that time was EJB, a standard environment for developing and deploying server-side components.[31] Although such

components must be implemented in Java, they are deployed at a tier behind components inter-acting directly with a client. Typically, they support Web applications using Java Servlets and Java Server Pages. The EJB framework for server-side components is compelling, but EJBs must run in a container. Although there are open source alternatives to avoid the high cost of commercial EJB containers, the runtime environment necessary to support EJBs is expensive in terms of system re-sources and complexity, especially for a standalone HPAC deployment. Furthermore, EJB servers were not well established at that time.

Web/HTTP. In Section 3.1 we compare rich GUIs vs Web applications with regard to the client. A Web application would require Web application services to produce the HTML pages, which leads to consideration of providing all HPAC services via HTTP, addressable via URLs. Note these considerations occurred before the advent of "Web Services" based on the Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL), and Universal Description Dis-covery and Integration (UDDI) protocols. Consequently, providing services across HTTP would differ little from doing so at the socket level, for HTTP provides no structure for object or data exchange. Thus, the same disadvantages hold for HTTP and socket services. The client would be responsible for adhering to an exchange protocol and data formats (parsing and formatting) HPAC would define and impose.

Web Services. Were the HPAC re-engineering effort to begin today, SOAP, WSDL, and UDDI, collectively Web Services, would warrant careful consideration as an alternative to CORBA for supporting clients independent of language and platform. However, there are several issues in comparing Web Services to CORBA that prevent the former from being the instant choice. First and most obvious is the level of maturity of the technologies. Web Service implementations have appeared only within the last calendar year, compared to decades of work on CORBA.

Another consideration is performance. CORBA must translate data to language- and platform-neutral *binary* formats. SOAP exchanges data as Extensible Markup Language (XML) documents, a verbose ASCII format requiring significant parsing time.

Finally, CORBA enforces strong typing. That is, objects and the parameters to their methods are defined in interfaces or structures in IDL. Any attempt at an incorrect use of an object will be caught at compile time. Conversely, type mismatches with SOAP are caught at runtime rather than compile time. There are benefits of both approaches, but with strong typing an unexpected problem detected at runtime is less likely, an important issue for system maintainability and reli-ability.

CORBA Alternatives. Through version 2.2, the CORBA standard imposes a limitation on inter-action with distributed objects passed as parameters to object methods. Pass-by-value semantics are not supported for remote objects, even when passed as an argument for an "in" (as opposed to "out" or "inout") parameter. Instead, remote objects always have pass-by-reference semantics. Note this is not the case for objects of defined types (e.g., structures and unions). They are passed by value.

An object passed by reference to a server method may be manipulated in that method. Any changes in the object's state remain when the method returns. It is necessary that clients under-stand this limitation.

With version 2.3 of the CORBA/IIOP (Internet Inter-Orb Protocol) specification, this problem was addressed.[32] Moreover, with the RMI over IIOP technology in Java, RMI clients can access

CORBA services directly, without first compiling CORBA IDL to generate the interface classes.[33] This is also a compelling capability, as it allows developers of HPAC Java client components to focus on Java remote interfaces and avoid maintenance of CORBA IDL sources.

Nonetheless, HPAC employs CORBA 2.0 for a couple of reasons. First, experiments early in the development process with RMI over IIOP were not encouraging. Specifically, services defined as `java. rmi. Remote` interfaces and compiled using tools available at that time resulted in IDL definitions not easily implementable in a language other than Java. Since HPAC must be extensible and allow the addition of source model implementations, defining interfaces in IDL proved more effective. Second, forcing clients to a 2.3 compliant object request broker (ORB) was considered too restrictive due to the unavailability of such an ORB when the new HPAC development effort began.

3.5.2 Server Architecture Elements

Factory Pattern. Server objects registered with a naming service can be accessed by multiple clients simultaneously, requiring the server objects to be mindful of multi-threading issues. Given the complexity of correctly accounting for multiple threads, a single threading model for source model servers is warranted. HPAC uses the Factory design pattern to achieve single threading for CORBA services.

As illustrated in the collaboration diagram of Fig. 8, a singleton *factory server* for each incident source model object is registered with the naming service. It provides a method, `getInstance()`, which returns a *per-client* instance of the model server object. Each server instance exists to serve a single client for a single *session*.
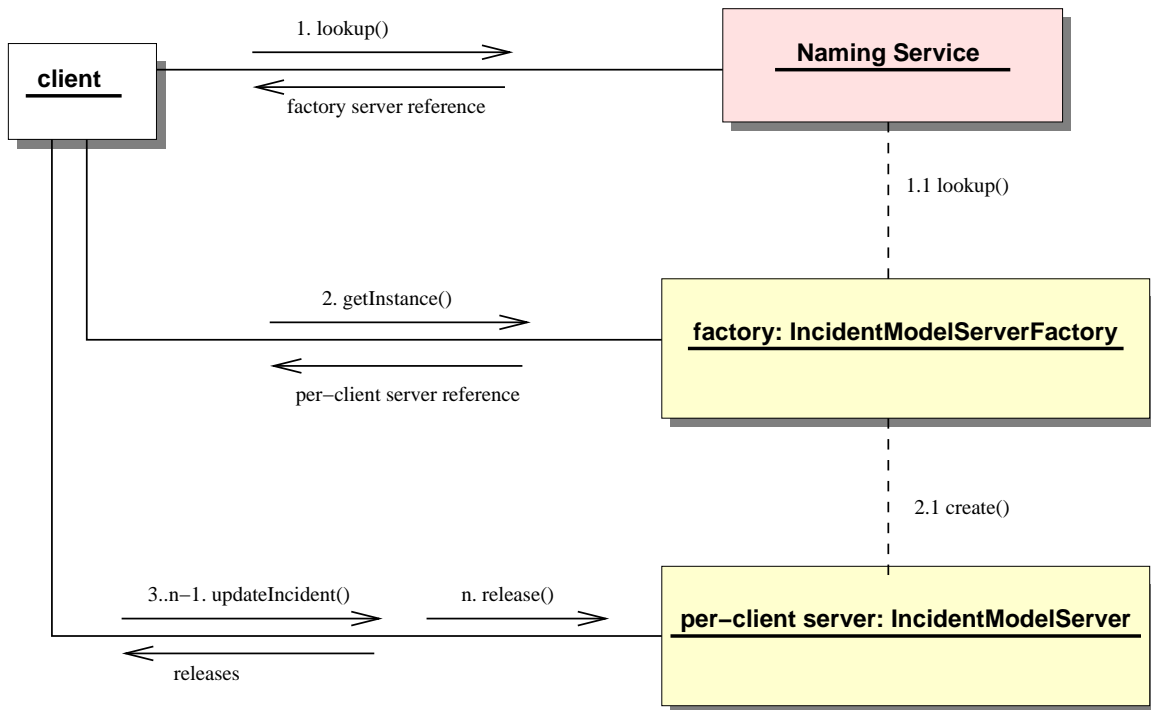

Fig. 8: Factory pattern collaborations.

20

Connectionless, Stateless Model.   The duration of the connection (or session) between a client and server across the network has a direct impact on performance and robustness. HPAC assumes a *connectionless*, *stateless* model where the session duration is assumed to be relatively short. That is, clients do not obtain server references and hold them between groups of calls for related activity. Rather, a client obtains a per-client server, performs needed work, and then releases the server. When the same services are needed again later, another server instance is obtained. Thus, clients neither hold connections, nor do they depend upon any state that existed in the previous server instance. This eliminates the need for clients to check the state of connections and, if necessary, repair or reestablish them.

Although reconnecting each time a service is needed adds additional communication, this approach has been effective due to the large latency between successive service calls from client objects. Continual or frequent service access would call for client objects to hold onto service references.

Model Server Abstract Base Class.   As described in Section 3.3.2, HPAC makes heavy use of abstract base classes to provide default behavior for framework interfaces. An abstract base class implementing the IDL-defined model server interface is provided as part of the incident model server framework. This serves several purposes. It relieves the model server implementations from implementing the same functionality, standardizes behavior among the implementations, and provides a level at which new requirements can be met and design revisions can be realized without undue disruption for server developers.

IDL Structures and Interfaces.   HPAC services involve several parameters of various data types and structures that must be passed from the client to the server object. How this is designed into the CORBA server objects has a huge bearing on performance.

One approach for passing this data is to define accessor methods for an interface by which the client sends the parameter values. Another is to define IDL structures encapsulating the parameters and then pass the structures and other parameter data in methods which operate on them. The performance implication arises from the fact that each method call involves communication across the network from client to server. Thus, assigning structure fields on the client and then sending all the parameter data en masse results in fewer communications. Although the amount of data to exchange is the same, the total communication time is reduced by avoiding accessor methods on server objects defined with IDL interfaces. By way of analogy, using structures is like sending a fully loaded truck with all cargo as opposed to sending multiple trucks, each with a single piece of cargo.

For example, the `calculateWithUrban()` method of the `DispersionCalculator` interface is defined with the IDL below.

```
boolean
calculateWithUrban(
    in IncidentTList          incidents,
    in AnyList                model_incidents,
    in WeatherT               weather,
    in LimitT                 limits,
    in OptionsT               options,
    in FlagsT                 flags,
    inout TemporalDomainT      temporal_domain,
    inout SpatialDomainT       spatial_domain,
```

```
        in float                    max_time_step,
        in float                    output_interval,
        in string                   udm_params,
        in string                   uwm_params
        )
        raises ( ScipuffException );
```

This method has 12 parameters. With one exception, all the non-primitive parameters are defined as IDL structures. The IDL definitions for two of them are listed below. Were structures not used at all, the `DispersionCalculator` interface would require setter methods for each of the 12 parameters, which would require 12 separate method calls or communications from the client to the `DispersionCalculator` server once the parameter objects were populated. The `OptionsT` structure defined below has 14 fields. Were it instead defined as an interface and implemented as a server object with setter methods for each field, 14 communications would be necessary to populate it alone. Consequently, HPAC service definitions make heavy use of IDL structures for parameter values.

```
    ...
    struct LimitT
      {
      long          fMaxPuffs;
      long          fMaxGridCellsPerSurface;
      long          fMaxMetHorizSize;
      };
    ...
    struct OptionsT
      {
      long          fVertGridTurbBL;
      long          fGridResolution;
      long          fSubstrateIndex;
      float         fTurbDiffAvgTime;
      float         fMinPuffMass;
      float         fMinAdaptiveGridSize;
      float         fTropVertVelVariance;
      float         fTropAvgDissRate;
      float         fTropVertScalelength;
      float         fCalmTurb;
      float         fCalmScaleLength;
      float         fDosageCalcHeight;
      float         fSamplerOutputInterval;
      string        fSamplerFile;
      };
```

3.5.3   File References

When deployed in a true client-server mode, there are many files that may need to be transferred from the client host to the server host for processing by source model services and the T&D engine. Examples include weather files, terrain and land cover files, and release files. These files are produced or obtained by the user and stored on the client host.

Since the sizes of the files to transfer can vary greatly, a "one size fits all" approach to uploading them to the server host is ineffective. The ultimate determining factor in the size of most files is

the spatial domain of the project. A global domain will result in very large files on the order of megabytes, potentially even tens of megabytes. With a typical spatial domain, weather files such as surface and upper profiles will have sizes on the order of tens of kilobytes.

Smaller files can be transferred by passing their contents in parameters to CORBA object methods. However, tests against the ORB provided with the J2SE (version 1.3.1) revealed that once the size of a parameter approaches ten megabytes, CORBA failures can occur. Thus, a separate upload mechanism is used for larger files. Specifically, files of size greater than 256 kilobytes are transferred via an explicit upload to the server host, whereas the contents of smaller files are passed in the method parameter.

Components and objects on the client and server side should not be burdened with explicit handling of the transfer mechanism. A file transfer mechanism transparent to all components is described below.

File References.   An IDL union, `FileReferenceT` is defined to encapsulate the file and the representation of its transfer to the server host. `FileReferenceT` and its supporting structures are listed below.

```
...
struct FileContentsT
  {
  string               fClientPath;
  sequence<octet>      fContents;
  }; // FileContentsT
...
struct FileServerPathT
  {
  string               fClientPath;
  string               fServerPath;
  }; // FileServerPathT
...
union FileReferenceT  switch( long )
  {
  case FR_CLIENT_PATH:
    string             fClientPath;

  case FR_CONTENTS:
    FileContentsT      fContents;

  case FR_DEFERRED:
    string             fDeferred;

  case FR_SERVER_PATH:
    FileServerPathT    fServerPath;

  case FR_URL:
    string             fURL;
  }; // FileReferenceT
```

A file reference can be in one of the five states specified by the union discriminators. A *deferred* reference is essentially empty or unspecified. A state of *client path* indicates the path to the file on

the client host has been specified, but the file has yet to be resolved or transferred to the server. *URL* is similar to *client path* with the file specified as a URL instead of a path on the client host.

The remaining two states represent the file reference after it has been resolved with respect to the server. The process of resolving file references is described below. For *contents*, the contents of a file originating on the client host are stored in the `FileReferenceT` object. A *server path* state means one of two things. Either the file has been explicitly transferred or uploaded from the client host to the server host, or the deployment is standalone, meaning the client and server hosts are the same and thus the path to the file on the server host is made equivalent to the client path. An additional service, the `FileServer` exists solely to support uploads of files during file reference resolution.

Resolving File References on the Client.   Prior to calling CORBA server object methods, a client component must first resolve `FileReferenceT` objects in the *client path* state. A utility class, `File-ReferenceTMgr` provides this and other methods for managing file references, a utility class being necessary since `FileReferenceT` is defined as an IDL union. `FileReferenceTMgr` resolves references by first invoking Java's reflection mechanism to recursively search the specified object and all its contained objects (i.e., field members) for instances of `FileReferenceT`. This reflective search calls the `java.lang.Class.getFields()` method, which processes only public field members. Thus, all `FileReferenceT` objects to be resolved must be tagged with `public` access, which also means all the owning objects in the hierarchy must also be public. By specifying the access tag, the developer of a class can control whether or not a `FileReferenceT` member is automatically resolved by `FileReferenceTMgr`.

Each `FileReferenceT` object in the *client path* state is resolved in one of three ways. First, if HPAC is running in standalone mode, the client file path is copied as the server file path, and the state is set to *server path*. Second, if the file is smaller than 256 kilobytes in size, the file's contents are read and stored in the file reference, the state being set to *contents*. Finally, files larger than 256 kilobytes are uploaded to the server host via the `FileServer` object. The state of the reference object is set to *server path*, the path of the uploaded file on the server stored in the reference.

Resolving File References on the Server.   The client-resolved `FileReferenceT` object is passed as a parameter (or a field in the structure hierarchy of a parameter) of a CORBA object method. Once received by the server object, the file must be obtained according to the state of the reference. A state of *deferred* is effectively a null or empty file reference and is treated as such. If the state is *server path*, the file already exists on the server host, and no other action is necessary other than to note the file's location on the server host. A state of either *contents* or *URL* means the file must be built, either by storing the contents passed with the reference object in the case of *contents* or by retrieving and storing the contents from a *URL*.

3.5.4   Supporting Standalone Deployment

In the original concept for the new HPAC architecture, standalone deployment was to be supported by merely co-locating client and server components on the same host. Certainly, this works, but CORBA's translations to language- and platform-neutral formats, in addition to time required to connect to network ports, requires some time, even when all communication occurs on the same host and never sees a physical network medium. The performance decrement was enough to warrant a solution to expedite communication between client and server objects on a standalone host.

The eventual solution was possible due to the choice of the Java Naming and Directory Interface (JNDI)[34] for all server object registration and lookups. JNDI implementations provided by the J2SE on top of the COSNaming service and an RMI registry are used for registering CORBA and RMI services, respectively. For standalone mode, a JNDI implementation based on the `java.util.Hashtable` class is used for CORBA and RMI server objects.

By registering server objects in a `Hashtable` object in the same JVM (i.e., operating system process), client objects retrieve references to the server objects themselves instead of stubs used for remote access. Thus, CORBA transport is bypassed and replaced with direct calls to the server objects. The result is significantly improved performance in standalone mode.

It should be noted that one kind of server object could not be deployed in this manner, specifically the CORBA wrapper for the T&D engine. Since the engine is written in non-reentrant Fortran, each per-client instance must always live in its own JVM process.

3.5.5 Supporting Thin Client Deployment

Use of URLs for file references as described in Section 3.4.1 solves the major problem in deploying HPAC as a *thin* or *lite* client, obtaining necessary data transparently to the using components. In this case, *thin* refers not only to the required capabilities of the client host but also suggests that no pre-installation of HPAC components is necessary. Rather, all that is necessary for running HPAC as a thin client is a platform for which a J2SE implementation is available and a Web browser.

The HPAC client GUI is a Java *application*, as opposed to an *applet*. This means it runs in a JVM executing as a process in the operating system and not within a Web browser. Requirements call for HPAC to be launched from a Web browser with no prior HPAC installation. Java WebStart™[35] is the mechanism by which this requirement is met. WebStart uses the Java Network Launching Protocol (JNLP) to download or update an application described in an XML document and linked in a Web page. The WebStart launcher must first be installed on the client host, but once installed, any application defined by a JNLP description document can be downloaded and executed on the client host. Again, no prior installation of HPAC components is necessary.

3.6 INTEGRATION

As stated above, integration with other tools and systems is the principle (if not transcending) goal behind the new HPAC architecture. Not only do organizations differ in their planned uses of HPAC, but systems with which HPAC must integrate have varying architectures and means of tool integration. In hopes of supporting all these needs, four levels of access to HPAC are provided as illustrated in Fig. 9:

- HPACtool library,
- detailed CORBA services,
- IHPACServer, and
- reusable Java beans and components.

3.6.1 HPACtool Library

At the lowest level of HPAC access is the HPACtool API and library.[36] HPACtool is a Windows C/Fortran library for SCIPUFF, the T&D engine. It also provides plot and output generation and
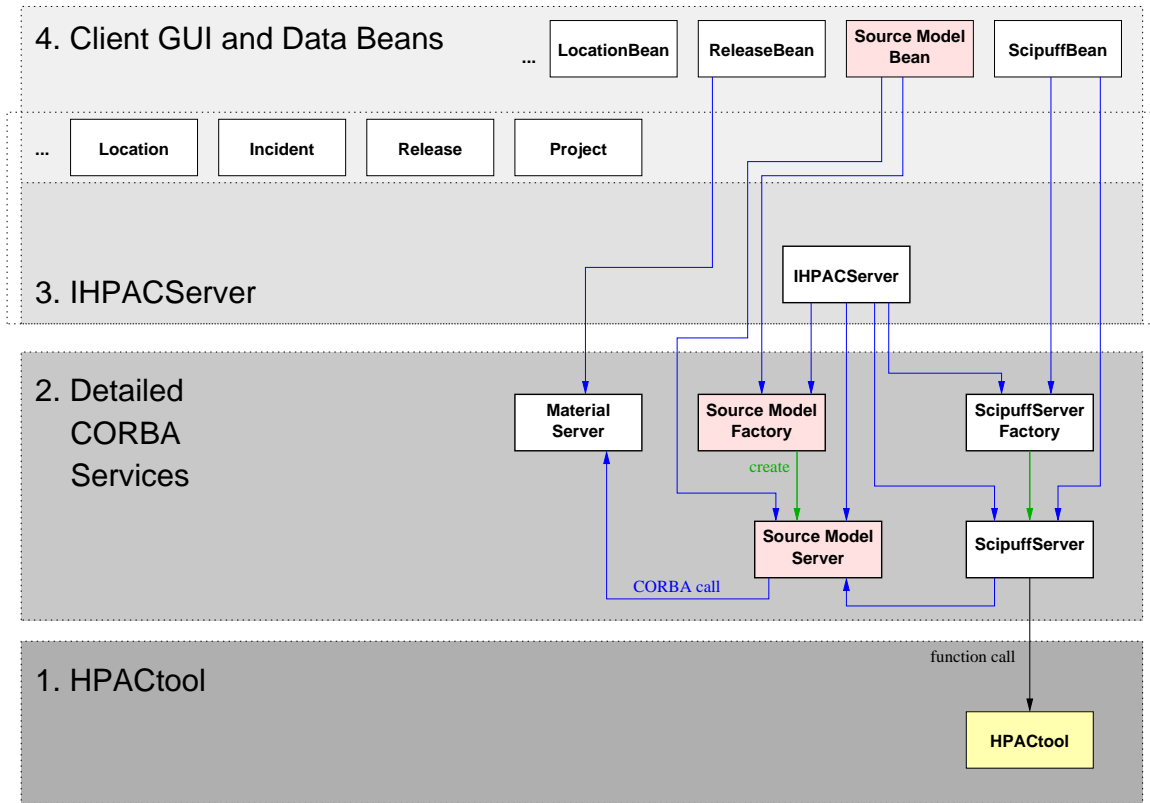
Fig. 9: Levels of HPAC access.

calls for effects and consequence computations, reading auxiliary data (e.g., population and land cover) as necessary. HPACtool provides no support for incident source models, instead taking detailed release and material descriptions as input. With the HPACtool library, organizations can link in or tightly integrate HPAC T&D calculations with their tools.

3.6.2   Detailed CORBA Services

The next level is a set of CORBA services providing HPACtool functionality as well as incident source model calculations, a material server, and radiological computations. Other systems may access HPAC's transport and dispersion calculation functionality as an object on the network instead of requiring them to link in a library on a specific platform. Further, incident source models are available to convert operational incident definitions into detailed release descriptions for input to the T&D engine.

At this level, all the detailed parameters, flags, and options of HPACtool are exposed to client objects. Values for all HPACtool parameters must be provided by the client in server object calls, but this gives the client complete control over the calculation process. This is the level at which HPAC client components access HPAC services.

3.6.3 IHPACServer

Another CORBA interface is provided by IHPACServer, a collection of server components hiding many of the detailed parameters. IHPACServer objects pass default values for many parameters to detailed objects by reusing Java data beans and components built for the HPAC client application and providing default calculation parameter values. It also makes use of the Java plotting components to obtain plot data and output from the HPACtool. However, IHPACServer's simpler interface comes at the cost of limiting access to some detailed parameters.

Further, IHPACServer is designed to be easy for developers of client components to invoke. Whereas detailed CORBA services make heavy use of IDL structures for optimal performance as described in Section 3.5.2, IHPACServer is defined with IDL interfaces exclusively, and all interactions are with remote objects. Clients with limited network bandwidth or performance concerns may be forced to detailed services.

3.6.4 Reusable Java Components

Almost all components in the HPAC client application are designed to be reused, both within HPAC and by other tools. HPAC's assortment of data and GUI beans represent the highest level at which other tools can access HPAC. An organization can effectively re-implement the HPAC client GUI by assembling the beans and components into a new application. However, for other tools to take advantage of this level of access to HPAC, they must be implemented in Java or at least have the ability to bridge to Java components.

4 SUMMARY

HPAC has been re-engineered from a Windows application with tight binding between computation and the GUI to a new distributed object architecture. The key goals of this new architecture are platform portability, extensibility, deployment flexibility, client-server operations, easy integration with other systems, and support for a new map-based GUI.

Selection of Java as the development and runtime language and environment is the major factor in achieving each of the goals, platform portability in particular. Portability is further enforced by allowing only Java components in the client. All native (C, C++, Fortran) code sits behind server components.

Extensibility is achieved via Java's dynamic binding and class loading capabilities and a design by interface approach. These techniques form the basis for building pluggable frameworks throughout HPAC, most notably for incident source model servers and client beans and map displays. Instead of Java's binary serialization mechanism, a framework based on Java properties files and the `java.util.Properties` class is used for (de)serialization of all objects in HPAC. The result is an easy mechanism for new implementations to be plugged into HPAC.

HPAC must support deployment on a standalone host, as a *heavy* client in client-server mode with data stored on the client but calculations performed on the server host, and as a *thin* client with data and calculations on the server host. The principle architectural element supporting deployment flexibility is the use of URLs for all file references. This works very well for sequentially-read files, but direct-access files must be served across the network. Large files are better handled by processing the data in a server object and sending only necessary data to the client. A necessary aspect of deployment flexibility is the use of runtime configuration files. Thin client deployment is provided via the Java WebStart™ facility.

Although there were many choices for the object distribution mechanism, CORBA was chosen to support HPAC client server operation. More specifically, HPAC uses version 2.0 of the CORBA standard and does not assume support for pass-by-value method arguments, thereby making it easier for clients to access HPAC.

Another issue in supporting client server operation is references to files stored on the client host. They must be transferred to the server host when needed. Rather than use a single transfer mechanism for all files regardless of size, HPAC employs a mechanism that passes the contents of small files in method parameters, but large files are first uploaded to the server with a reference to the uploaded file passed in a method parameter. Execution in standalone mode is expedited by executing most server objects in the same JVM as client objects, thereby bypassing CORBA object transport.

For integration with other tools and systems, HPAC provides four levels of access, beginning with the HPACtool API and Windows library providing transport and dispersion calculations as well as effects computations and output generation. However, no access to source models is provided in HPACtool. The next level up is a set of detailed CORBA services including incident source model servers. Detailed control over calculations is provided at this level.

An additional set of CORBA services is provided in IHPACServer. Here the detailed calculation parameters are defaulted. Although the interface is simpler, access to detailed parameters is limited, and server objects require more network communication. Finally, the reusable Java data and GUI components developed for the HPAC client application are available to any other application. However, a Java environment is necessary to make use of this highest level of access to HPAC.

References

[1] SCIPUFF Dispersion Model,
   http://www.titan.com/appliedtech/Pages/TRT/pages/scipuff/scipuff.htm.

[2] R. I. Sykes, C. P. Cerasoli and D. S. Henn, "The Representation of Dynamic Flow Effects in a Lagrangian Puff Dispersion Model", J. Haz. Mat., 64, 223-247, 1999.

[3] R. I. Sykes and R. S. Gabruk, "A Second-Order Closure Model for the Effect of Averaging Time on Turbulent Plume Dispersion", J. Appl. Met., 36, 165-184, 1997.

[4] R. I. Sykes, D. S. Henn, S. F. Parker and R. S. Gabruk, "SCIPUFF - A Generalized Hazard Dispersion Model", Ninth Joint Conference on the Applications of Air Pollution Meteorology with A&WMA, American Met. Soc., 1996.

[5] DTRA Technology Development Directorate,
   http://www.dtra.mil/td/td_index.html.

[6] High Level Architecture Interface Specification, Version 1.3, U.S. Department of Defense, April 1998.

[7] Integrated Target Planning Toolset, http://www.itpts.com/.

[8] Warfighter's Simulation (WARSIM),
   http://stricom.army.mil/PRODUCTS/WARSIM/.

[9] V. Maciejewski and J. Zukowski, "Developing Rich User Interfaces for Software Services", Spidertop Technical Report TR101, Spidertop, Inc., October 2001.

[10] H. Rosen, H. Muller, S. Violet, Rich Clients for Web Services, Presentation at JavaOne 2001, June, 2001,
http://servlet.java.sun.com/javaone/conf/sessions/2734/0-sf2001.jsp,
http://java.sun.com/products/jfc/tsc/articles/javaOne2001/2734/

[11] Tim Lindholm and Frank Yellin, The Java Virtual Machine Specification Second Edition, Addison Wesley, 1999, http://java.sun.com/docs/books/vmspec/index.html.

[12] Brent Welch, Practical Programming in Tcl and Tk, Third Edition, Prentice Hall, 1999.

[13] Eric F. Johnson, Graphical Applications with Tcl and Tk, Second Edition, M&T Books, 1997.

[14] Python Website, http://www.python.org.

[15] The Fast Light Toolkit Home Page, http://www.fltk.org.

[16] "Object-Oriented Programming Concepts: A Primer", Java Tutorial,
http://java.sun.com/docs/books/tutorial/java/concepts/index.html.

[17] Jeffrey R. Shapiro, "Sleeping with the Primitives", NetworkWorldFusion, November 2001,
http://www.nwfusion.com/newsletters/java/2001/01123457.html.

[18] Grady Booch, Object Oriented Design with Applications, Benjamin/Cummings, 1991.

[19] The Java HotSpot Virtual Machine, Technical White Paper, Sun Microsystems Inc., Palo Alto, California, 2001.

[20] The Java Performance Report, The JavaLobby,
http://www.javalobby.org/fr/html/frm/javalobby/features/jpr/.

[21] P. Przemyslaw and B.N. Bershad, "Dynamic Binding for an Extensible System", Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, WA, pp. 201-212, October 1996.

[22] Java™ Core Reflection API and Specification,
http://java.sun.com/products/jdk/1.1/docs/guide/reflection/spec/java-reflectionTOC.doc.html.

[23] BioJava Design Overview, http://www.biojava.org/docs/.

[24] Rob Shecter, "Design by Interface", Dr. Dobb's Journal, February 1999,
http://www.ddj.com/documents/s=906/ddj9902j/9902j.htm.

[25] Mark Grand, Patterns in Java Volume 1, pp. 377-383, Wiley, 1998.

[26] OpenMap™ Open Systems Mapping Technology,
http://openmap.bbn.com

[27] T. Berners-Lee, R. Fielding, L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax", RFC 2396, August 1998,
http://www.ietf.org/rfc/rfc2396.txt.

[28] TCP/IP, Webopedia, http://www.webopedia.com/TERM/T/TCP_IP.html.

[29] Microsoft COM for Solaris,
http://www.microsoft.com/com/resources/solaris.asp.

[30] JavaBeans Bridge for ActiveX,
http://java.sun.com/products/javabeans/software/bridge/.

[31] Enterprise JavaBeans™ Technology,
http://java.sun.com/products/ejb/index.html.

[32] The Common Object Request Broker: Architecture and Specification, Revision 2.3.1, Object Management Group, Inc., October 1999.

[33] Java™ RMI Over IIOP,
    http://java.sun.com/products/rmi-iiop/index.html.

[34] Java Naming and Directory Interface (JNDI),
    http://java.sun.com/products/jndi/index.html.

[35] Java WebStart™,
    http://java.sun.com/products/javawebstart/.

[36] Stephen F. Parker, The HPAC Application Programming Interface HPAC Version 4.0 (Draft), Titan Research and Technology, Titan Corp., Princeton, NJ, December 2000.

[37] A. Nguyen-Tuong, S.J. Chapin, A.S. Grimshaw, C. Viles, "Using Reflection for Flexibility and Extensibility in a Metacomputing Environment", University of Virginia Technical Report CS-98-33, 1998.

INTERNAL DISTRIBUTION

  1. J. C. Gehin
  2. R. T. Goeltz
3-7. R. W. Lee
  8. R. H. Morris
  9. T. E. Potok
10. J. S. Tolliver
11. B. A. Worley
12. ORNL Central Research Library
13. ORNL Laboratory Records

EXTERNAL DISTRIBUTION

13. Thomas A. Mazzola, Northrop Grumman Information Technologies, 6940 S. Kings Hwy, STE 210, Alexandria, VA 22310
14. Ronald Meris, Defense Threat Reduction Agency, 6810 Telegraph Rd, Alexandria, VA 22310